LAWRENCE
LIVERMORE
NATIONAL
LABORATORY

LLNL-CONF-681383

Investigation of Portable Event-Based Monte Carlo Transport Using the NVIDIA Thrust Library

R. C. Bleile, P. S. Brantley, S. A. Dawson, M. J. O'Brien, H. Childs

January 27, 2016

2016 American Nuclear Society Annual Meeting
New Orleans, LA, United States
June 12, 2016 through June 16, 2016

Disclaimer

Investigation of Portable Event-Based Monte Carlo Transport Using the NVIDIA Thrust Library

Ryan C. Bleile[*,†], Patrick S. Brantley[*], Shawn A. Dawson[*], Matthew J. O'Brien[*], Hank Childs[†]

[*]*Lawrence Livermore National Laboratory, P.O. Box 808, Livermore, CA 94551*
[†]*Department of Computer and Information Science, University of Oregon, Eugene, OR 97403*
bleile1@llnl.gov, brantley1@llnl.gov, dawson6@llnl.gov, obrien20@llnl.gov, hank@uoregon.edu

INTRODUCTION

Power consumption considerations are driving future high performance computing platforms toward many-core computing architectures. Los Alamos National Laboratory's Trinity machine, available in 2016, will use both Intel Xeon Haswell processors and Intel Xeon Phi Knights Landing many integrated core (MIC) architecture coprocessors. Lawrence Livermore National Laboratory's Sierra machine, available in 2018, will use an IBM PowerPC architecture along with Nvidia graphics processing unit (GPU) architecture accelerators. These different advanced architectures make the computing landscape in upcoming years complex.

Traditional approaches to Monte Carlo transport do not work efficiently on these new computing platforms. MIC architectures require vectorization to operate efficiently, and vectorization is difficult to achieve in Monte Carlo transport. GPU architectures require additional code to explicitly use the hardware, requiring significant code changes or hardware specific branches in the source code. A significant challenge for Monte Carlo transport projects is to simultaneously support within a single source code base efficient simulations for both the current generation of architectures and the different advanced computing architectures.

In order to address these challenges, two important changes are typically required: a new algorithmic approach for solving Monte Carlo transport, and explicit use of hardware specific software. In this paper, we describe initial research investigations of an event-based Monte Carlo transport algorithm [1] implemented using the Nvidia Thrust library [2] on a GPU for a Monte Carlo test code. The event-based algorithm targets many-core architectures by increasing SIMD (single instruction multiple data) parallelism, while Thrust potentially provides portable performance by allowing one source code base to compile code targeted for both CPUs and GPUs.

HISTORY-BASED APPROACH

For this research, we began with the ALPSMC Monte Carlo test code [3] that models particle transport in a one-dimensional planar geometry binary stochastic medium. The ALPSMC code was originally implemented in C++ using a standard history-based Monte Carlo transport algorithm, as shown in Alg. 1. This approach follows a single particle from creation until it is absorbed or leaked. Parallelism is easily achieved by parallelizing over particle histories (foreach loop on Line 1), with each thread working independently on a single particle at a time. This approach uses a MIMD (multiple instruction multiple data) parallelism scheme, as opposed to vectorization.

Algorithm 1: History-based Monte Carlo algorithm

```
1  foreach particle history do
2      generate particle from boundary condition or source
3      while particle not escaped or absorbed do
4          sample distance to collision in material
5          sample distance to material interface
6          compute distance to cell boundary
7          select minimum distance, move particle, and perform event
8          if particle escaped spatial domain then
9              update leakage tally
10             end particle history
11         if particle absorbed then
12             update absorption tally
13             end particle history
```

EVENT-BASED APPROACH

Previous researchers [4, 5, 6] have noted that the use of an event-based Monte Carlo particle transport algorithm [1] may be beneficial for GPU or vector-based architectures. We investigated this idea through the event-based algorithm shown in Alg. 2 as a way to potentially optimize performance on GPU and vector-type architectures. In event-based particle tracking, the individual events can be treated by a series of data parallel operations. The data parallel model matches the vector and GPU hardware with an emphasis on performing the same operations on many pieces of data at one time through SIMD parallelism.

Thrust

Thrust is a C++ header library using a STL-like template interface [2]. Thrust provides a number of parallel algorithms and data structures designed to provide access to GPU computing without needing to write CUDA code [7]. Additionally, Thrust provides backend capabilities allowing these algorithms and data structures to target different devices, including CPUs with OpenMP threads. This design was used for studying portable performance techniques with Thrust, providing a method of maintaining only one source code.

Thrust algorithms are used for implementing procedures across all particles in a batch. These algorithms perform operations such as the data parallel map, reduce, gather, scatter, or scan operations defined in [8]. Each of these operations can be performed in a data parallel way.

Algorithm 2: Event-based Monte Carlo algorithm

```
1  foreach batch of particle histories (fits in memory constraint) do
2      generate all particles in batch from boundary condition or source
3      determine next event for all particles (collision, material interface crossing, cell boundary crossing)
4      while particles remaining in batch do
5          foreach event E in (collision, material interface crossing, cell boundary crossing) do
6              identify all particles whose next event is E
7              perform event E for identified particles and determine next event for these particles
8          if particle escaped spatial domain then
9              update leakage tally
10         if particle absorbed then
11             update absorption tally
12         delete particles absorbed or leaked
```

Thrust also provides data types that can be used to manage memory for GPU devices. The thrust::device_vector and thrust::host_vector data structures operate similarly to a C++ std::vector but with automatic memory copying between host and devices whenever necessary. These data types allow for simple memory management schemes that work on both GPU and CPU based architectures.

Algorithm Detail

An event-based algorithm focuses on performing data parallel operations across all particles undergoing the same event. Additional overhead is needed to find the grouping of particles that will be operated on and to determine an access pattern for the particles. This reorganization stage can be costly and is not directly related to solving the transport problem.

Thrust provides permutation iterators that allow for the unaligned access of data elements according to an index map. Using this iterator scheme, data elements do not need to be copied into new locations for each operation. This approach comes at the cost of performing non-contiguous memory accesses for reading and writing the information.

In order to perform an event operation on particles using this scheme, a series of data parallel operations is used to establish the correct index mapping for the permutation iterator. This scheme is defined as follows and describes in detail lines six and seven of Alg. 2:

Step 1: thrust::transform — Fill out a stencil map of 1's and 0's of all particles doing event E (where each particle whose next event is E will get a 1 in the stencil map at its index location)

Step 2: thrust::reduce — Count the number of elements labeled 1 in the stencil (determines the number of particles that will perform event E)

Step 3: Check if the number of elements is greater than 0 (check if any particles are performing event E)

Step 4: thrust::exclusive_scan — generate indices for index mapping from stencil map (indices for each particle performing event E)

Step 5: Allocate a new map of appropriate size (map to hold indices for all particles performing event E)

Step 6: Scatter indexes from scan into new index map (reduces the exclusive_scan generated indices into the map that holds only enough for particles performing event E)

Step 7: Use new index map in permutation_iterator loops over all particles (combining the index map with the permutation iterator allows loops over all particles to operate only on the particles selected in the index map)

Related Work

Other researchers have recently performed related work implementing event-based algorithms on GPUs [4, 5, 6]. Comparing and contrasting our event-based method to theirs shows differences both in our approaches, target problem, and the way chosen to create data parallelism. While their research first studied the idea of event-based Monte Carlo on GPUs, our research studies event-based Monte Carlo in the light of the portable performance platform of Thrust as well as an algorithm designed around performing only data parallel operations.

The research performed by Bergmann et. al [6] with WARP is most closely related to our work. We both implemented an algorithm that performs a series of operations on the GPU in a single loop over particles still active. Additionally, we both implemented a remapping vector to identify particles for a task without copying them. While in WARP the remap vector is then sorted so that particles will most likely be close in memory to other particles undergoing the same operation, in our implementation only particles that will undergo the same event are called in one kernel. Additionally, our remap vector does no sorting and only identifies whether particles will undergo a specific event. In this way, we implemented our event-based model as a series of data parallel calls with Thrust. Each of our calls has almost no divergence except when deciding if a particle leaked or was absorbed.

The work done by Liu et al [5] within ARCHER to test an event-based model differs from our work algorithmically. Liu et. al did use Thrust for some operations but chose to make the important kernels CUDA only. Additionally, Liu's algorithm used a double while loop focusing all attention on one event and then switching to the other, making their code have two events to consider.

The work done by Nelson [4] is related in the attempt to use and optimize performance on a GPU, but event-based methods were not considered in that work.

IMPLEMENTATIONS

We implemented the event-based version of ALPSMC using both the Nvidia CUDA programming model [7] explicitly and the Nvidia C++ Thrust library [2]. The Thrust implementation of ALPSMC utilizes data parallel operations and Thrust data types for managing memory. The same Thrust event-based implementation can be compiled with either CUDA for use on GPUs or OpenMP for use on CPUs, enabling portability to different platforms. In the native CUDA implementation of ALPSMC, we found it useful to continue to use Thrust algorithms in building various maps. ALPSMC is implemented using double precision floating point numbers throughout. The Thrust and CUDA implementations of ALPSMC give physics results identical to the original history-based implementation.

The CUDA implementations for this study matched the algorithm in the Thrust implementations. The differences in performance come from the capabilities that native CUDA programming provide that cannot be accomplished with Thrust. Using CUDA directly enables more fine-grained control at the kernel level and enables important access to different memory spaces such as GPU shared memory. The CUDA implementation includes a scheduling algorithm to optimize the number of active threads on the GPU for each kernel call. Additionally, the CUDA implementation includes the use of the different available memory spaces, such as constant and shared memory. For example, Monte Carlo particles were initially allocated in GPU global memory and then copied to shared memory for all operations within a kernel. All problem constants such as cross sections and mean chord length values were placed in GPU constant memory. These optimizations under certain conditions can have a significant impact on the performance of a GPU kernel.

NUMERICAL RESULTS

We performed scaling studies in which we varied the number of Monte Carlo particle histories (problem size) and the implementation methodology (Thrust or CUDA). The results presented are for Case 1a [3] with a spatial domain of 10 cm. We also examined the differences in performance on three different computer platforms. The Rzgpu computer has Intel Xeon Westmere-EP 2.8 GHz host cores with Nvidia Tesla M2070 GPU device accelerators. The Max computer has Intel Sandy Bridge 2.6 GHz host cores with Nvidia Tesla K20X GPU device accelerators. The Tesla K20X GPU has improved double precision performance over the Tesla M2070. The Rzhasgpu computer has Intel Xeon Haswell 3.2 GHz host cores with Nvidia Tesla K80 GPU device accelerators. We did not use a multiple GPU implementation and were therefore only able to utilize approximately half of the computational power of the Nvidia Tesla K80s.

Our first study aimed to identify the speedups of our event-based algorithm when compared to the initial serial history-based implementation. We computed speedups over a serial calculation by dividing the wall clock time of a serial run of the history-based version of ALPSMC on the host core of the given machine by the wall clock time of the event-based version of ALPSMC running on both a single host CPU and the GPU device. The speedups obtained on each computing platform are shown in Table I.

TABLE I: ALPSMC Event-Based Monte Carlo GPU Speedups Over Serial History-Based Application

	Number Particle Histories		
	10^6	10^7	10^8
CUDA (K20X)	5.90	11.88	11.91
CUDA (1/2 K80)	4.88	10.49	10.51
CUDA (M2070)	3.96	6.05	6.05
Thrust (K20X)	2.11	2.60	2.60
Thrust (1/2 K80)	1.77	2.17	2.17
Thrust (M2070)	1.42	1.64	1.63
Thrust OpenMP Event	2.54	2.15	2.22

For this test, the Thrust implementation produces speedups ranging from approximately 1.4 to 2.6. Therefore, while the Thrust library potentially provides an approach to obtaining a portable implementation, it does not produce the significant speedups we would expect on the GPU hardware. For this test, the speedups obtained using the CUDA implementation of the event-based algorithm are significantly larger than those obtained using the Thrust implementation by up to over a factor of four. We attribute this improved performance to the fact that CUDA offers more control over the memory spaces available on the GPU (e.g. shared memory) when operating on large kernels that perform multiple read/write actions. Thrust does not offer such flexibility and manages the memory allocation internally. We conclude based on these preliminary investigations that a direct CUDA implementation is more efficient than a Thrust implementation for event-based Monte Carlo. Also, the speedups on the Max platform (Tesla K20X GPU) are larger than on the Rzgpu platform (Tesla M2070 GPU) by up to a factor of approximately two, presumably a result of the improved double precision performance of the K20X. Furthermore, the Rzhasgpu platform (Tesla K80 GPU) shows similar performance to the Max platform (K20X GPU); we can assume around twice the performance were we to modify the research code to fully utilize all of the available K80 hardware.

The same Thrust event-based code implementation was compiled with OpenMP for use on the host CPU, demonstrating the portability of the Thrust implementation. The scaling study was repeated on Rzhasgpu's Intel Xeon Haswell CPUs with OpenMP using 16 threads/cores. The CPU performs similar to the GPU when Thrust is used to gain parallelism, with speedups of approximately 2.2. Using 16 OpenMP threads, we would expect a significantly larger speedup for Monte Carlo particle transport. Since the same code base is used with Thrust on both the CPU and the GPU, we can see the potential that exists for a single code base on multiple platforms. For this particular example, however, significantly higher performance is achieved using the native choice of the CUDA event-based implementation for the GPU.

We performed a second more extensive scaling study varying the number of particle histories for the Thrust and CUDA event-based versions and the serial history-based version on Rzhasgpu (Tesla K80 GPU). The results of the scaling study are shown in Figure 1. We can see that both the Thrust and CUDA event-based versions have significantly higher overhead than the serial history-based version at low numbers of particle histories. But at a higher number of particle histories (starting at approximately 10^5 particle histories), the event-based versions of the code begin outperforming the serial history-based versions. We also observe that the performance gains of the CUDA version over the Thrust version start to become significant at higher numbers of particle histories.

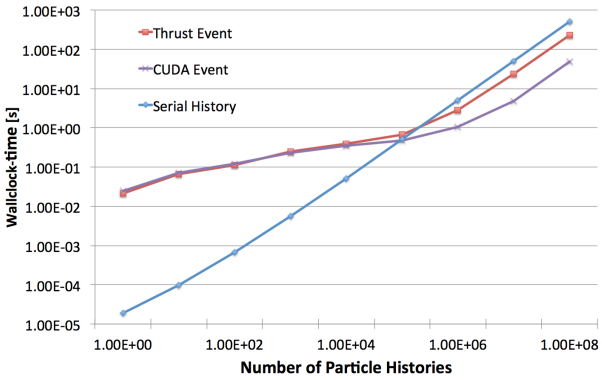

Fig. 1: Log-log plot of wall clock time versus number of particles for different versions of the code

The linear behavior observed at the higher particle history counts is a result of the particle batching scheme we used to avoid exhausting GPU device memory. Once the batching begins, we no longer gain any additional performance increases. At that point, the only performance improvement possible would be to process a greater number of particle histories, and that number is hardware dependent.

CONCLUSIONS

We described preliminary investigations of portable event-based Monte Carlo algorithms implemented using the Nvidia Thrust library in a research Monte Carlo test code. We found that an explicit CUDA implementation of an event-based Monte Carlo algorithm performed significantly more efficiently than a Thrust implementation on GPU platforms, most likely as a result of additional flexibility in access to different memory spaces on the GPU. Additionally, we showed that on GPU platforms and at large enough problem sizes the event-based implementations perform more efficiently than the serial history-based implementation running on the host CPU.

While investigating this problem, we also discovered that the performance of the event-based algorithm is affected by what tallies are being used. A zonal scalar flux tally requires atomic operations that significantly impacted the performance of the code, in some cases producing slowdowns instead of speedups. We decided to remove the tally in order to focus on the effectiveness of the event-based algorithm. Future work will be required to research more effective ways of handling such tallies. Additionally, we would like to consider new ways for optimizing both the Thrust and CUDA versions, in order to see how much performance we have yet to achieve.

The potential trade-off between portability and performance was demonstrated in this investigation. Thrust provides both CPU and GPU versions of the code in one code base, but it does so at a cost. We discovered approximately a factor of 2-5 performance difference between Thrust and CUDA on each GPU platform we tested. Thrust provides a tool to access the GPUs with less effort and specialization, but it does so by giving up fine grained control where extra performance can be found for this application. Future work will be required to determine whether this performance differential between Thrust and CUDA can be reduced for event-based Monte Carlo transport.

ACKNOWLEDGMENTS

This work was performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344. Funding was provided by the LLNL Livermore Graduate Scholar Program.

REFERENCES

1. F. B. BROWN and W. R. MARTIN, "Monte Carlo Methods for Radiation Transport Analysis on Vector Computers," *Progress in Nuclear Energy*, **14**, 269–299 (1984).
2. "Thrust Web Site," (2014), https://developer.nvidia.com/Thrust.
3. P. S. BRANTLEY, "A Benchmark Comparison of Monte Carlo Particle Transport Algorithms for Binary Stochastic Mixtures," *Journal of Quantitative Spectroscopy and Radiative Transfer*, **112**, 599–618 (2011).
4. A. G. NELSON, *Monte Carlo Methods for Neutron Transport on Graphics Processing Units Using CUDA*, M.S. Thesis, The Pennsylvania State University (2009).
5. T. LIU, X. DU, W. JI, X. G. XU, and F. B. BROWN, "A Comparative Study of History-Based Versus Vectorized Monte Carlo Methods in the GPU/CUDA Environment for a Simple Neutron Eigenvalue Problem," in "Proceedings of Supercomputing in Nuclear Applications and Monte Carlo (SNA+MC)," Paris France (October 27-31, 2013 2013).
6. R. M. BERGMANN and J. L. VUJIC, "Algorithmic Choices in WARP - A Framework for Continuous Energy Monte Carlo Neutron Transport in General 3D Geometries on GPUs," *Annals of Nuclear Energy*, **77**, 176–193 (2015).
7. "CUDA Web Site," (2014), http://www.nvidia.com/object/cuda_home_new.html.
8. G. E. BLELLOCH, *Vector models for data-parallel computing*, vol. 356, MIT press Cambridge (1990).